# Intel® Thread Checker Getting Started Guide

**Intel® Thread Checker detects data races, deadlocks, stalls, and other threading issues. It can detect the potential for these errors even if the error does not occur during an analysis session. Use Thread Checker to filter out specific types of diagnostics, identify critical source locations, and see tips on ways to improve the robustness of your parallel software.**

**To quickly start using Thread Checker, print this short guide and walk through the example provided.**



## Overview

This guide presents a threaded code example and teaches you how to use Intel® Thread Checker to identify and handle threading-related issues. After completing this guide, you should be ready to analyze and repair your own code using Thread Checker.

# 1. Build the Sample Code

The `Primes` sample code identifies and tallies the prime numbers in the range from one to 1,000. Using the Windows* threading API's, multiple threads perform the work. However, the threads simultaneously access the same memory location, causing potential data races. As a result, this program may generate incorrect results.

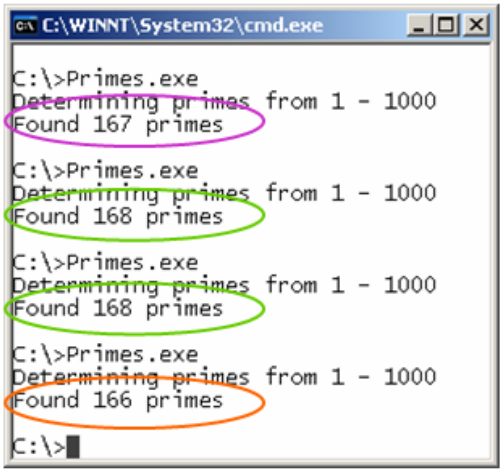
### To build the code:

1. Open the `Primes.dsw` project workspace file in Microsoft* Visual Studio. By default, this project is installed with the Intel® Thread Checker in: `C:\Program Files\Intel\VTune\tcheck\Samples\Primes`.
2. Build the `Primes.dsw` project.
   This sample project is set up to include the following options: `"/Zi"` to include symbols, `"/Od"` to disable optimization, linked `"/fixed:no"` to make code relocatable, and `"MDd"` or `"MTd"` to build with thread-safe run-time libraries. These options are required to enable Thread Checker to provide you with the most detailed information, including variable names and line numbers associated with errors.

***TIP***

*See Thread Checker online* **Help** *for more hints on options and building applications.*

If you run the `Primes.exe` program several times, you might see output such as the following:




What do you see? Different runs produce inconsistent results! The correct result is "`Found 168 primes`". In this case, it is relatively easy to see that there is a threading inconsistency. In larger programs, a threading inconsistency can be much harder to see.

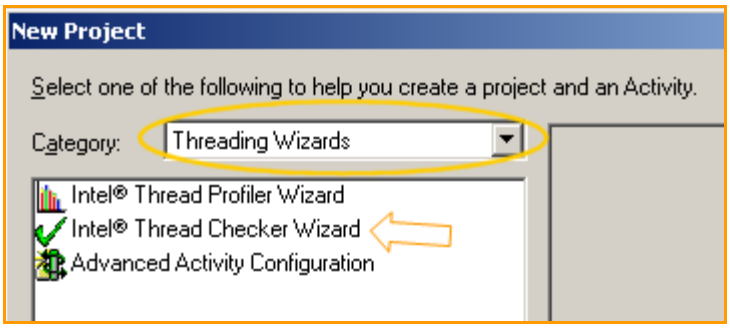
In the next section, you will use Thread Checker to collect data to help you locate the threading inconsistency.

# 2. Collect Data

You will now use the **Intel® Thread Checker Wizard** to create a project and an Activity and collect data on the `Primes.exe` image you built.

### To collect Intel® Thread Checker data:

1. Double-click the **Intel® VTune™ Performance Analyzer** shortcut icon on your desktop to start the VTune™ Analyzer with the Intel® Thread Checker plug-in: 
2. In the **Easy Start** dialog box that opens, or from the main toolbar, click the **New Project** button  to open the **New Project** dialog box.
3. In the **Category** drop-down box, choose **Threading Wizards** and select  **Intel® Thread Checker Wizard** as shown:

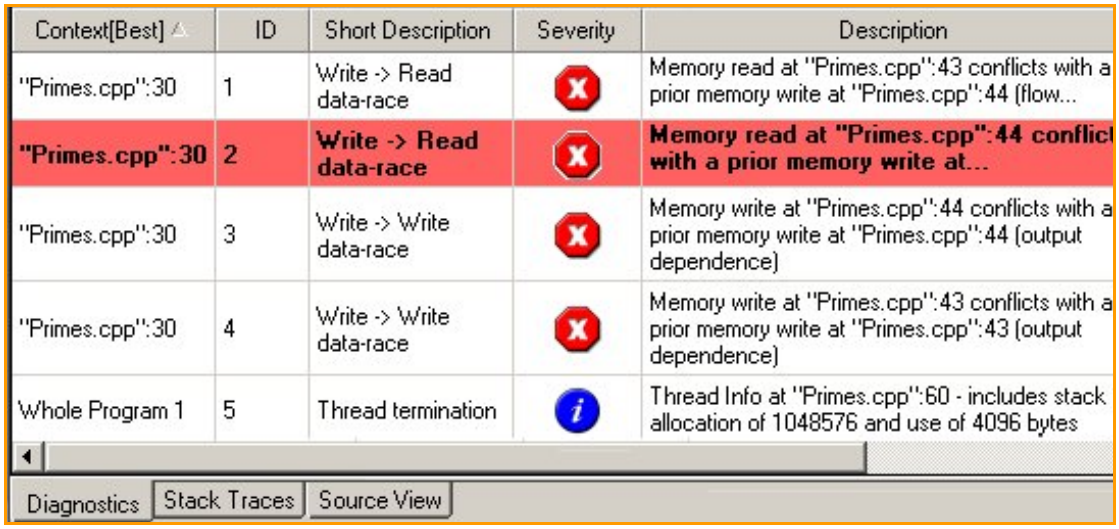



4. Enter a **Project Name**, for example, *`PrimesProject`*.
   Thread Checker fills in a default for the **Project Location** directory that you can change if required.
5. Click **OK**. The **Intel® Thread Checker Wizard** opens.
6. Under **Launch an application**, click **[…]** to navigate to the `Primes.exe` you built.
   By default, Thread Checker fills in the **Working directory** with the executable's directory. For all other options, you can use default values.
7. Click **Finish** to complete the wizard.
   Thread Checker instruments your application, executes it, collects data, and displays the results in the **Diagnostics** list.

In the next section, you will use Thread Checker to identify threading issues.

# 3. Analyze Results and Correct the Code

After completing the wizard, you should see results similar to the following:



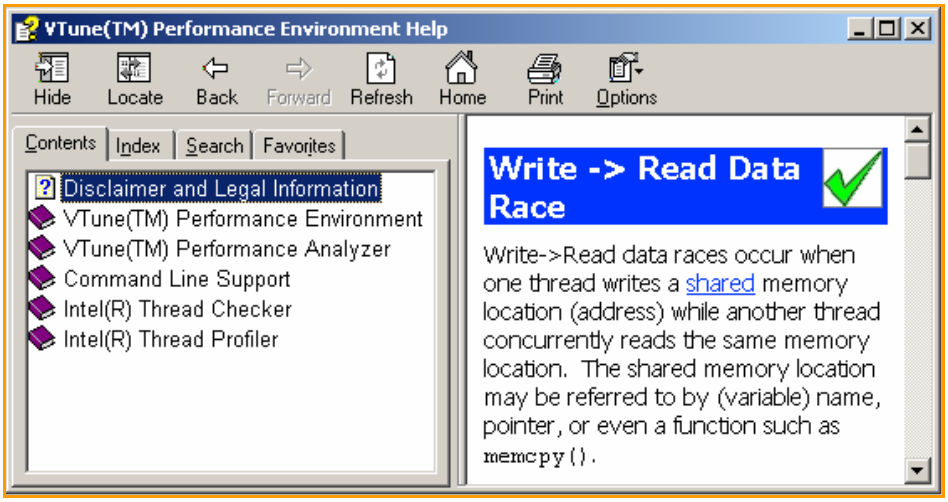

Congratulations! You are now ready to analyze diagnostics and correct threading inconsistencies in the application.

**TIP**

*If you do not see results, open* **Help** > **Search** *and search for* "**Troubleshooting Thread Checker"** *for possible causes and solutions.*

### To analyze diagnostics:

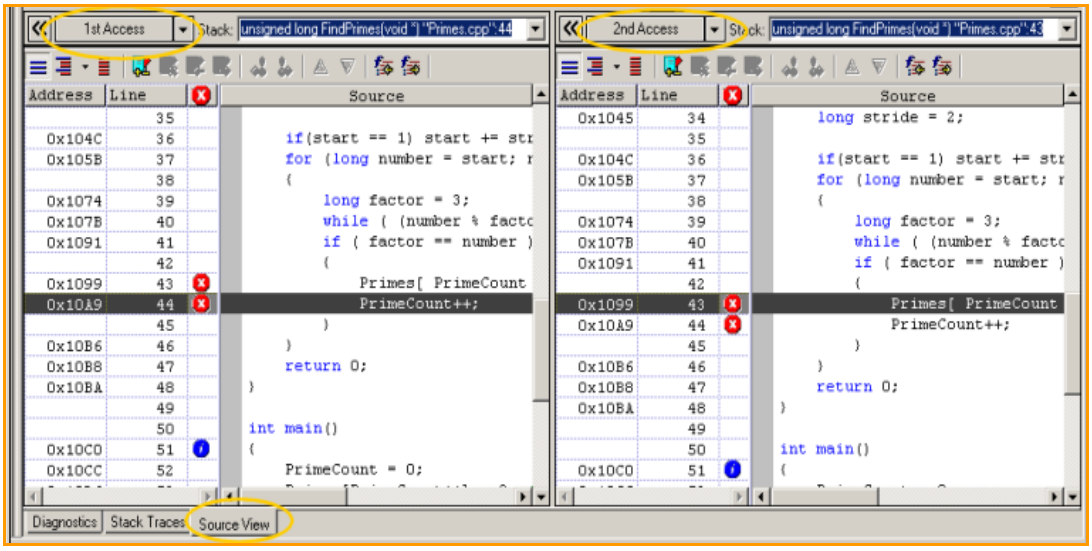
1. Look at the first diagnostic in the list, highlighted in red and identified by **ID 1**. Thread Checker identified a **Write -> Read data-race** error. This error, indicated by a red stoplight icon,  , is caused by shared memory variable accesses that are not properly synchronized**.**
2. Right-click the first row in the diagnostic list and select **Diagnostic Help** from the pop-up menu. A **Help** topic opens, showing explanations of the diagnostic, its causes, and possible solutions to help you fix your code.



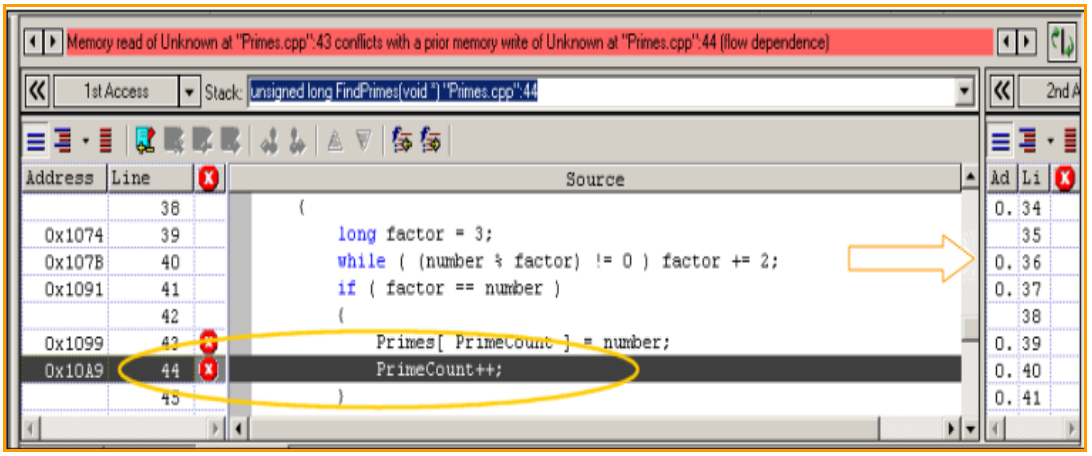
**Write->Read** data races occur when one thread writes a shared memory location (address) while another thread concurrently reads the same memory location, causing a conflict.

### To locate the conflict in the source:

1. Click the **Source View** Tab. The **1st Access** pane shows the source location where the previous thread wrote to the shared memory location. The **2nd Access** pane shows the source location where the unsynchronized memory read of the shared memory location occurred:




2. You can get a closer look at each access by expanding the **Source Views** separately by expanding the panes:




### Resolve conflicts and test results

3. Try adding a `CRITICAL_SECTION` to correct the code. By adding a synchronization object to serialize the use of `Primes[]` and `PrimeCount` (lines 43 and 44), you can eliminate the inconsistent and incorrect effects of time sliced multi-threading.
4. To verify that the Write -> Read data-race identified by Thread Checker is indeed fixed, repeat the steps you followed above:

1. Build the modified code.
2. Collect data on the modified `Primes.exe` by pressing the **F5** key, or by clicking the Run Activity button  .
3. Analyze results.

How did you do? If you resolved all the data races, you should see results with only informational diagnostics indicated by a blue icon,  .

**HINT**

*View the code provided in* `PrimesFixed.cpp` *for a solution to the data race conflicts. You can also build this code using the* `PrimesFixed.dsw` *project and analyze the* `PrimesFixed.exe` *within Thread Checker to confirm that the data races are in fact fixed.*

## 4. Next Steps

To get the most out of Thread Checker, explore the following resources:

1. **Online Help** is the product's complete user's guide. Use **Help** to learn about additional features, including remote data collection and other advanced options.
   Open **Help** by pressing the **F1** key.
2. **Samples** provide additional code examples for you to explore. Use them to learn to identify and resolve other types of threading errors.
   Find code **Samples** in the `tcheck\Samples` folder. Read the associated ***Guide to Sample Code***, `CodeExamplesGuide.pdf`, available in the `tcheck\Doc` folder.
3. **Release Notes** include key product details. See the Release Notes for updated information on requirements, technical support, and known limitations.
   Open **Release Notes** from (for example) **Start > Programs > Intel® Software Development Tools > Intel® Thread Checker > Release Notes**.
4. **Frequently Asked Questions** (FAQ) provides details on Intel® Thread Checker Remote Data Collection on LINUX* systems.
   Open the FAQ from (for example)`/opt/intel/itt/tcheckFAQ.htm`.
5. **Intel® Thread Profiler**. After you check your code with Intel® Thread Checker, use the **Intel® Thread Profiler** to help you improve its performance.
   Find details about Thread Profiler and other Intel software development products at: http://www.intel.com/software/products/.

# Disclaimer and Legal Information